TESTE DE *SOFTWARE*

AULA 6

Prof.ª Maristela Weinfurter

CONVERSA INICIAL

Nesta abordagem, trataremos dos processos de teste de *software*, e utilizamos alguns exemplos para ilustrar várias maneiras de especificar casos de teste e critérios de saída. O teste dinâmico é baseado em técnicas de caixa preta, caixa branca e baseadas na experiência. Explicamos o que cada categoria significa e como escolher a mais adequada para a sua situação. Na contrapartida, também falamos sobre testes estáticos e suas características.

Ao resgatarmos todas as possibilidades de testes, encontramos várias formas de testes de *software*, orientado a objetos ou não, bem como orientado a outros paradigmas de desenvolvimento de *software*.

Dentre algumas técnicas importantes que encontramos para testes de *software*, estão: estruturais e funcionais. As estruturais (caixa branca) testam o código fonte, cada linha que for possível, fluxos básicos e alternativos do código.

Quando falamos em testes funcionais, encontramos testes de *stress*, execução, recuperação, operação, segurança, entre outros.

Ao pensarmos nos testes funcionais, também conhecidos como *caixa preta*, somos conduzidos à análise funcional do *software*, garantindo que os requisitos funcionem conforme foi especificado. Não nos importa aqui a forma como o código foi implementado; apenas inserimos dados e observamos os resultados para confrontá-los com os requisitos.

Dentre os testes funcionais mais conhecidos, temos os de requisitos, regressão, tratamento de erros, manual, de UI, de controle e de paralelismo.

Agora, vamos mergulhar nos testes tipo caixa preta e caixa branca.

TEMA 1 – TESTES ESTÁTICOS

O teste estático e a análise de produtos de trabalho, tanto documentação quanto código, contribuem de forma mensurável para aumentar a qualidade do produto. Vamos falar sobre esse tipo de teste estático em geral, bem como o processo específico envolvido, com suas atividades e funções que devem ser preenchidas. Destacaremos algumas técnicas e suas vantagens específicas, bem como os fatores que garantem o sucesso ao aplicá-las:

- Técnica subestimada: o teste estático (ou “análise estática”) pode ser executado em um ambiente baseado em ferramentas ou manualmente, e é uma técnica de teste frequentemente negligenciada. Enquanto o objeto

de teste para um teste dinâmico é um programa executável executado usando dados de teste, o teste estático pode ser executado em qualquer tipo de produto de trabalho que seja relevante para o desenvolvimento de um produto. O teste estático pode assumir a forma de exame minucioso por uma ou mais pessoas ou pode ser realizado usando ferramentas de análise apropriadas.

- Tudo é prevenção: o conceito subjacente do teste estático é a prevenção. Erros e outros desvios do plano são identificados antes que possam ter um efeito negativo durante o desenvolvimento posterior. Todos os produtos de trabalho relevantes têm qualidade assegurada antes que alguém faça qualquer trabalho adicional neles. Isso evita que produtos de trabalho provisórios defeituosos entrem no fluxo de desenvolvimento e causem falhas dispendiosas posteriormente.

Os testes estáticos, segundo Nook (2021), podem ser executados em várias situações e geralmente são parte integrante do processo de desenvolvimento, ou seja, os recursos de teste são disponibilizados no estágio de planejamento. Por exemplo, em indústrias críticas que exigem muita segurança, como aviação e medicina, a análise estática é um fator extremamente importante para garantir a qualidade do produto exigida.

1.1 O que analisar e testar

Assim como o próprio código, o desenvolvimento de *software* também produz muitos outros produtos de trabalho. A maioria dos documentos desempenha algum tipo de função no desenvolvimento posterior de um produto, tornando a qualidade de cada documento individual um fator importante na qualidade geral dos resultados.

Os produtos de trabalho que podem ser verificados usando análise estática incluem:

- especificações;
- requisitos de negócios;
- requisitos funcionais;
- requisitos não funcionais;
- requisitos de segurança.

Erros de especificação precisam ser encontrados antes de serem convertidos em código, e em projetos ágeis, épicos e histórias de usuários, eles estão sujeitos a testes estáticos.

Já os tipos de defeitos que a análise estática revela incluem:

- inconsistências;
- ambiguidades;
- contradições;
- lacunas;
- imprecisões;
- redundâncias.

Durante o processo de *design* de *software*, são criadas especificações de arquitetura e *design*, como, por exemplo, diagramas de classe, que podem ser testados estaticamente como o código do programa. O código que compõe um *site* também pode ser testado dessa maneira, utilizando-se a verificação automática de *links* e seus *sites* de destino correspondentes. O código pode ser verificado quanto à consistência com as diretrizes de programação do projeto.

De modo geral, o teste estático verifica se quaisquer padrões predefinidos foram respeitados nos documentos que estão sendo analisados. Também é aconselhável examinar e verificar todos os documentos, definições e ferramentas criadas durante o teste, como plano de teste, casos de teste, procedimento e *scripts* de teste e critérios de aceitação. Outros documentos e produtos de trabalho que podem ser verificados por testes estáticos incluem contratos, planos de projeto, cronogramas, orçamentos e manuais do usuário.

Além disso, os resultados dos testes estáticos também podem ser usados para melhorar o processo geral de desenvolvimento. Se certos tipos de defeitos ocorrerem repetidamente durante etapas específicas de desenvolvimento, isso é um sinal de que essa etapa requer investigação e, se necessário, otimização. Esse tipo de processo geralmente é acompanhado por treinamento adicional de pessoal.

1.2 Técnicas de testes estáticos

Habilidades mentais e analíticas humanas podem ser usadas para analisar e avaliar situações complexas por meio de uma investigação minuciosa dos documentos em questão. É essencial para o sucesso de tal análise que as

pessoas envolvidas entendam os documentos que estão lendo e compreendam as declarações e definições que eles contêm. Os testes estáticos geralmente são a única maneira eficaz de verificar a semântica da documentação e outros produtos de trabalho.

São várias técnicas de teste estático que diferem em sua profundidade, nos recursos que requerem (pessoas e tempo) e nos objetivos que perseguem. Algumas técnicas comuns de teste estático são detalhadas a seguir:

- A técnica de teste estático mais comum e importante é a revisão. Neste contexto, o termo *revisão* tem múltiplos significados. É usado para descrever a análise estática de produtos de trabalho, mas também como um termo geral para todas as técnicas de análise estática executadas por seres humanos.
- O termo *inspeção* também é usado para descrever um processo semelhante, embora seja frequentemente usado para significar a execução formal de um teste estático, bem como a coleta de métricas e dados de acordo com regras predefinidas.
- As revisões podem ser conduzidas informal ou formalmente. As revisões informais não seguem um processo predefinido e não há uma definição clara dos resultados pretendidos ou do que é registrado. Em contraste, o processo de revisão formal é predefinido, e os participantes documentam um conjunto planejado de resultados. O tipo de processo de revisão escolhido dependerá dos seguintes fatores:
  - O modelo de ciclo de vida de desenvolvimento: qual modelo está sendo usado?
  - Quais lugares no modelo e resultados provisórios são adequados para conduzir uma revisão?
  - A maturidade do processo de desenvolvimento: qual é a maturidade do processo e qual é a qualidade dos documentos a serem verificados?
  - A complexidade dos documentos a serem testados: qual saída mostra um alto grau de complexidade e, portanto, é adequada para uma revisão formal?
  - Requisitos legais ou regulamentares e/ou a necessidade de uma trilha de auditoria rastreável: quais regulamentos devem ser respeitados e quais provas de garantia de qualidade são necessárias?

- Os objetivos desejados de uma revisão também determinam que tipo de processo de revisão devemos escolher:
  - Focamos na detecção de defeitos ou na compreensão geral dos produtos de trabalho testados?
  - Os novos membros da equipe precisam se familiarizar com um documento específico revisando-o?
  - A revisão ajuda a equipe a chegar a uma decisão consensual?
  - O tipo de revisão que conduziremos dependerá de suas respostas a essas perguntas.

1.3 O processo de revisão

O processo de revisão dos produtos de trabalho compreende planejamento, início, preparação pelos participantes — que podem ser especialistas, revisores e inspetores —, comunicação e análise dos resultados, correção de falhas e relatórios. Veja a Figura 1.

Figura 1 – Atividades do processo de revisão

1.3.1 Planejamento: defina seus objetivos e escolha um tipo de avaliação

No estágio de planejamento, o gerenciamento — ou, mais precisamente, o gerenciamento do projeto — deve decidir quais documentos ou partes de documentos devem estar sujeitos a qual tipo de revisão. Esta decisão afeta as

funções que devem ser preenchidas, as atividades necessárias e, se necessário, o uso de listas de verificação. Precisamos decidir quais características de qualidade avaliar. O esforço estimado e o tempo para cada revisão também devem ser planejados. O gerente de projeto seleciona uma equipe de participantes devidamente qualificados e atribui a eles suas funções. O gerente de projeto também deve verificar com os autores dos artefatos/objetos de revisão se eles estão em um estado passível de revisão, ou seja, se chegaram a pelo menos uma conclusão parcial e estão o mais completos possível.

As revisões formais requerem critérios de entrada e saída predefinidos. Se o plano inclui critérios de entrada, devemos verificar se eles foram cumpridos antes de prosseguir com a revisão. Diferentes pontos de vista aumentam a eficácia. Verificar um documento de diferentes pontos de vista ou fazer com que indivíduos verifiquem aspectos específicos de um documento aumenta a eficácia do processo de revisão. Esses pontos de vista e/ou aspectos devem ser definidos na fase de planejamento. Não precisamos revisar um documento em sua totalidade. Muitas vezes, faz sentido selecionar as peças que contêm defeitos de alto risco ou amostras de peças que permitem tirar conclusões sobre a qualidade do documento como um todo. Se desejarmos realizar uma reunião de pré-revisão, precisamos decidir quando e onde ela será realizada.

1.3.2 Iniciando uma revisão

O pontapé inicial de um processo de revisão encontra-se no momento que os participantes recebem todos os materiais físicos e eletrônicos necessários. Isso pode ser um simples convite por escrito ou uma reunião de pré-revisão na qual a importância, o propósito e os objetivos da revisão são discutidos. Se os participantes ainda não estiverem familiarizados com a configuração do objeto de revisão, o início também pode ser usado para descrever brevemente como o objeto de revisão se encaixa em seu campo de aplicação.

Juntamente aos produtos de trabalho que serão revisados, os participantes da revisão requerem outros materiais. Isso inclui quaisquer documentos que ajudem a decidir se o que eles estão vendo é um desvio, uma falha/defeito ou uma declaração correta. Essas são as bases com as quais o objeto de revisão é verificado (por exemplo, casos de uso, documentos de *design*, diretrizes e padrões). Esses documentos são muitas vezes referidos como a “linha de base”. Critérios de teste adicionais (talvez na forma de listas de

verificação) ajudam a estruturar o procedimento. Se estivermos usando formulários para registrar suas descobertas, eles precisam ser distribuídos no início do processo.

Se estivermos realizando uma revisão formal, precisaremos verificar se os critérios de entrada foram atendidos. Caso contrário, a revisão deve ser cancelada. Isso economiza tempo que seria desperdiçado revisando produtos de trabalho “imaturos”.

1.3.3 Preparação de revisão individual

Uma revisão só pode ser bem-sucedida se todos os participantes estiverem bem-preparados, portanto, cada membro da equipe de revisão deve se preparar individualmente para a revisão.

Os revisores submetem o objeto da revisão a um intenso escrutínio, usando a documentação fornecida como base. Quaisquer defeitos potenciais, recomendações, perguntas ou comentários são anotados.

1.3.4 Comunicação e análise de problemas

Após a preparação individual, os resultados são reunidos e discutidos. Isso pode ocorrer em uma reunião de revisão ou, por exemplo, em um fórum *on-line* interno da empresa. Os potenciais desvios e defeitos encontrados pelos membros da equipe são discutidos e analisados. Devemos definir quem é responsável pela correção das falhas identificadas, como monitorar o progresso da correção e determinar se uma revisão de acompanhamento do documento corrigido é necessária ou necessária. As características de qualidade sob investigação são definidas durante o planejamento. Cada característica é avaliada e os resultados da análise são documentados.

A equipe de revisão deve fornecer uma recomendação sobre a aceitação do objeto de revisão:

a) Aceito sem alterações ou com pequenas alterações;
b) Revisão necessária devido a mudanças extensas;
c) Não aceito;
d) Corrigindo defeitos.

3.1.5 Atividades finais (correções e relatórios)

As atividades finais no processo de revisão são relatar suas descobertas e corrigir quaisquer defeitos ou inconsistências que a revisão tenha revelado. As atas de uma reunião de revisão geralmente contêm todas as informações necessárias, de modo que relatórios individuais não sejam necessários para defeitos que exijam a modificação do objeto de revisão. Em regra, o autor corrigirá quaisquer defeitos revelados pela crítica.

Além de redigir relatórios, quaisquer defeitos encontrados também podem ser comunicados diretamente à pessoa ou equipe responsável. No entanto, isso envolve boas habilidades interpessoais, pois ninguém gosta muito de falar sobre seus próprios erros.

Revisões formais envolvem mais trabalho. Em uma revisão formal, precisamos registrar o *status* atual de um defeito ou seu relatório. A alteração do *status* de um defeito só é possível com o consentimento do revisor responsável e os critérios de saída especificados. Avaliar os resultados de uma reunião de revisão formal vai nos ajudar a melhorar o processo de revisão e mantermos atualizadas as diretrizes e listas de verificação correspondentes. Isso requer a coleta e avaliação de métricas apropriadas. Os resultados de uma revisão variam consideravelmente dependendo do tipo de revisão e do grau de formalidade envolvido.

TEMA 2 – TESTES DE CAIXA PRETA: PARTICIONAMENTO DE EQUIVALÊNCIA E ANÁLISE DE VALORES E LIMITES

2.1 Noções básicas sobre o particionamento de classe de equivalência

É impossível testarmos todas as entradas possíveis em nosso *software*, então temos que agrupá-las em categorias que demonstrem o comportamento de toda uma classe de entradas. Sem dúvida, já o fazemos em nossos planos de teste, e a isso chamamos de *particionamento de classe de equivalência*. Ao considerá-lo de forma abstrata, aprendemos a identificar possíveis casos de teste de forma rápida e completa (Amey, 2022).

Por exemplo, para testarmos se uma caixa de texto pode lidar com entradas incluindo espaços, podemos usar a *string* “um dois”. Esta *string* é um exemplo de todas as *strings* possíveis que incluem um espaço entre as palavras.

Eles são todos equivalentes, então os particionamos juntos, e esse exemplo é nosso único teste para verificar todo o grupo.

As *strings* a seguir testam outras partições:

Tabela 1 – Exemplos de partições de equivalência para *strings*

Exemplo	Teste
String incluindo espaços	“hello world”
String incluindo caractere de acento	“Élan”
String incluindo letras maiúsculas	“Hello”
String com caracteres especiais	“;:!@£$%îèà{[('")}]”

Um exemplo clássico de testes realizados com particionamento de classe de equivalência são as alíquotas de impostos. Por exemplo, um imposto de 0% é aplicado até R$ 12.000,00, 20% até R$ 40.000,00 e 40% acima desse valor. Nesse caso, as partições são claramente indicadas nos requisitos e nós devemos escolher um valor para testar dentro das faixas de imposto de 0%, 20% e 40% para um total de três testes, cobrindo todas as três partições válidas.

Esses não são os únicos casos de teste que utilizamos, é claro. Ainda temos as validações com partições inválidas, consideradas testes de casos de erro, e os limites entre diferentes partições, para as quais usamos a análise de valor de limite. A execução de testes de partição de equivalência não cobre todas as condições, mas identifica os casos que devem ser incluídos. O exemplo de imposto listou as possíveis partições na especificação, mas, geralmente por conta de sorte, poderemos obter o suficiente para limites de partição explicitamente definidos, porém teremos que tentar identificá-los.

Dentro de diferentes estados de um aplicativo, por exemplo, temos:

a) usuários que já são membros;
b) usuários registrados, mas sem senha;
c) novos usuários que não efetuaram login.

Para diferentes opções ativadas:

a) entradas válidas e inválidas;
b) valores numéricos funcionalmente diferentes;
c) maiores de 18 contra menores de 18 anos;

d) pedidos caros o suficiente para obter frete grátis.

E ainda para diferentes tipos de dados de entrada:

a) moedas;
b) fusos horários;
c) idiomas.

Como podemos ver, existem muitas partições possíveis em muitas variáveis diferentes. Precisaremos analisar nosso aplicativo para encontrar as partições relevantes. Depois de identificadas as diferentes partições equivalentes, escreveremos casos de teste para exercitar cada uma delas. Assim como na identificação de cenários de pior caso, muitas vezes devemos combinar partições de teste para reduzir o número de casos de teste, como para esta *string*: **Olá ;:!@£$%îèà{[('')}]**

Se funcionar, testamos caracteres acentuados, alguns caracteres especiais, letras maiúsculas e *strings* com um espaço simultaneamente. Tais combinações reduzem o número total de testes.

Por outro lado, se essa *string* falhar, saberemos qual aspecto causou a falha. Quando a chance de falha dos testes é maior, por exemplo, em código novo, vale a pena separar as diferentes condições de teste, mesmo que isso signifique rodar mais testes individuais. Da mesma forma, combinar casos pode economizar tempo com testes automatizados, mas a depuração é mais fácil se as condições forem separadas. No entanto, se não conseguirmos executar testes de regressão manualmente, eles serão lentos e provavelmente passarão. Nesse caso, combinar condições de teste pode acelerar o processo.

Devemos buscar por partições de acordo com a implementação que é invisível para o usuário. Por exemplo, inteiros podem ser codificados em 2 bytes até 65.535, e 4 bytes, acima desse valor. Essas são duas partições diferentes, e embora os requisitos não as distingam, a implementação o faz. Aplicando o princípio de particionamento por equivalência, podemos gerar casos de teste para entradas comuns, como *strings* e arquivos, porém devemos considerar os limites entre as partições. Isso é coberto pela análise de valor limite.

Dentro de uma partição, nem todos os valores são igualmente importantes. Os erros são muito mais prováveis nos limites das partições onde os erros *off-by-one* podem estar presentes, por exemplo, se *maior-que-ou-igual* for usado em vez de *igual*. A ideia é bastante simples: quando há um limite,

precisamos testar os valores mais baixos e mais altos para garantir que a divisão esteja no lugar certo. Assim como nas partições de equivalência, os limites podem ser explicitamente listados nas especificações ou podem ser aspectos implícitos de como um recurso foi implementado.

Exemplos de limites explícitos são:

a) Alíquotas: os valores até X mil são tributados em uma alíquota, e os valores de X a Y mil são tributados em outra.
b) Idades: usuários abaixo de 13 anos são banidos, usuários entre 13 e 18 anos obtêm uma conta infantil e usuários com mais de 18 anos obtêm uma conta adulta.
c) Senhas: o comprimento das senhas deve ter mais de oito caracteres.

Exemplos de limites implícitos dependerão dos detalhes de implementação do seu produto, mas procure valores e listas que são implementadas de uma maneira até certo ponto e de forma diferente para outros valores. Por exemplo, inteiros de tamanhos diferentes podem ser armazenados de forma diferente. A implementação tentará ocultar esses detalhes, mas podem ocorrer problemas. Os valores para limites inteiros são bastante simples. Para o exemplo de senha anterior, por exemplo, testaremos se o aplicativo rejeita senhas com sete caracteres e aceita senhas com oito caracteres. Para valores numéricos não inteiros, o valor logo acima do limite precisa levar em conta a precisão. Se seus primeiros R$ 10.000,00 são isentos de impostos, precisamos testar o que acontece com R$ 10.000,01, um centavo acima disso. Dado que não podemos reivindicar 20% sobre um centavo, o que acontece com o arredondamento? Isso também faz parte da análise de limites. Se a especificação do recurso não for precisa sobre o comportamento, se deve ser arredondado para cima ou para baixo, por exemplo, temos que adicioná-lo.

O exemplo de idade também possui valores não inteiros. Aos 12 anos e 364 dias, os usuários são rejeitados; devemos testar esse valor, assim como 13 anos e 0 dias. A idade é medida com mais precisão do que os dias? Espero que não! Mas isso também precisa ser explicitamente declarado na especificação.

A análise de valor de limite pode ser de dois valores, conforme descrito anteriormente, ou de três valores, também conhecida *como análise de limite total*. Para isso, testamos os valores *um abaixo*, *igual a* e *um acima do limite*. Assim, para a senha de oito caracteres, tentaríamos senhas de sete, oito e nove

caracteres. Pessoalmente, não consigo ver uma senha de nove caracteres falhando se uma senha de oito caracteres foi aceita corretamente. No entanto, a análise de limite de três valores faz parte do teste de teoria, então eu seria negligente se deixasse de mencioná-la. Além dos limites entre os valores, precisamos verificar como os valores interagirão uns com os outros.

TEMA 3 – TESTES DE CAIXA PRETA: TABELA DE DECISÃO, TESTE DE TRANSIÇÃO DE ESTADOS E TESTE DE CASO DE USO

3.1 Tabelas de decisão

Para capturarmos interações complexas entre variáveis dependentes e o comportamento do sistema, pode ser útil escrever as possibilidades em uma tabela. Isso fornece uma base para escrever casos de teste e garante que todas as condições sejam cobertas. Ao expandir as variáveis de maneira sistemática, confirmamos que não perdeu nenhuma combinação.

Consideraremos como exemplo um aplicativo da *web* com suporte básico ou avançado, dependendo do sistema operacional e do navegador da *web* no qual é executado. O modo avançado não substitui o modo básico; alguns usuários podem escolher o modo básico mesmo que o modo avançado esteja disponível.

Para a especificação, temos a seguinte afirmação:

- O Chrome oferece suporte aos modos básico e avançado no Windows e macOS
- O Edge suporta apenas o modo básico no macOS
- O Safari suporta apenas o modo básico e apenas no macOS
- O Edge oferece suporte aos modos básico e avançado no Windows

Esses requisitos cobrem todos os casos possíveis? Eles cobram, embora isso não seja imediatamente óbvio. Podemos tornar essas condições mais claras escrevendo-as, o que também nos mostra quantos testes precisamos no total:

Tabela 2 – Uma tabela de decisão para suporte de aplicativos em diferentes sistemas operacionais e navegadores da *web*

Sistema Operacional	Navegador	Modo do Aplicativo	Suportado?
Windows	Chrome	Básico	Sim
Windows	Chrome	Avançado	Sim
Windows	Edge	Básico	Sim
Windows	Edge	Avançado	Sim
Windows	Safari	Básico	Não
Windows	Safari	Avançado	Não
Mac	Chrome	Básico	Sim
Mac	Chrome	Avançado	Sim
Mac	Edge	Básico	Sim
Mac	Edge	Avançado	Não
Mac	Safari	Básico	Sim
Mac	Safari	Avançado	Não

Temos dois sistemas operacionais, três navegadores da *web* e dois modos de aplicativo, totalizando: 2 x 3 x 2 = 12 casos. Mesmo que alguns arranjos não sejam suportados, precisamos testá-los para garantir que eles falham de maneira controlada. Escrever a tabela inteira leva um pouco de tempo, mas tem a vantagem de listar explicitamente cada caso de teste.

Para evitar redundância, podemos recolher as linhas que não dependem de uma variável. Por exemplo, nas duas primeiras linhas da tabela anterior, o Chrome no Windows oferece suporte aos modos de aplicativo básico e avançado:

Tabela 3 – Um trecho de uma tabela de decisão expandida

Sistema Operacional	Navegador	Modo do Aplicativo	Suportado?
Windows	Chrome	Básico	Sim
Windows	Chrome	Avançado	Sim

Podemos reescrever isso para mostrar que o Chrome no Windows é compatível independentemente do modo de aplicativo:

Tabela 4 – Um trecho de uma tabela de decisão recolhida

Sistema Operacional	Navegador	Modo do Aplicativo	Suportado?
Windows	Chrome	-	Sim

A remoção de toda a redundância da tabela anterior fornece a seguinte tabela de decisão recolhida:

Tabela 5 – Uma tabela de decisão recolhida para suporte a aplicativos

Sistema Operacional	Navegador	Modo do Aplicativo	Suportado?
-	Chrome	-	Sim
-	Edge	Básico	Sim
Windows	Safari	-	Não
Windows	Edge	Avançado	Sim
Mac	Edge	Avançado	Não
Mac	Safari	Básico	Sim
Mac	Safari	Avançado	Não

Observe que a primeira linha indica que ambos os sistemas operacionais suportam o Chrome em ambos os modos de aplicativo. Quatro linhas se reduziram a uma. Comece com uma coluna primeiro e reduza-a o máximo possível antes de prosseguir; caso contrário, podemos eliminar as sobreposições.

Embora a tabela de decisão reduzida seja mais organizada, ela não mostra mais os testes explícitos que precisamos executar, portanto, a tabela de decisão expandida pode ser mais simples de escrever e mais útil na prática.

A análise de valor limite e a partição de equivalência complementam as tabelas de decisão. O particionamento por equivalência identifica as variáveis e seus diferentes valores, e então devemos determinar quais são dependentes e

independentes umas das outras. As tabelas de decisão expandem a lista completa de possíveis casos de teste e podemos, se necessário, identificar os limites entre eles executando a análise de valor de limite.

Onde várias variáveis interagem, pode ser útil exibir visualmente as dependências. Os gráficos de causa e efeito são uma maneira de conseguir isso, conforme descrito a seguir.

3.2 Teste de transição de estados

Muitos aplicativos existirão em diferentes estados durante o processamento. Mesmo aplicativos sem estado sem armazenamento persistente têm máquinas de estado transiente conforme as conexões são solicitadas e confirmadas, por exemplo, ou processam mensagens recebidas e enviam respostas. Uma parte essencial do teste de caixa branca é identificar e testar todos esses estados, as transições entre eles e os erros que podem ocorrer.

Alguns estados podem ser evidentes na especificação, como usuários que inseriram seu endereço de *e-mail*, mas ainda não o confirmaram ou escolheram uma senha e ainda não fizeram *login*. Esses estados e suas transições já serão cobertos como parte da caixa preta teste.

No entanto, muitos outros estados podem não ser óbvios para os usuários, que precisam ser descobertos e compreendidos como parte do teste de caixa branca. Um trecho de código pode aceitar solicitações da *web* recebidas, mas depois movê-las para diferentes encadeamentos ou filas para processamento. Ao trabalhar com os desenvolvedores, precisamos mapear todas essas sequências para verificar as transições entre cada estado. Isso inclui transições que levam a estados de erro, que muitas vezes não são aparentes nas interações usuais do usuário.

Tendo identificado os estados, precisamos escrever testes para verificar cada um deles. Isso exigirá que visitemos todos os estados e atravessemos todas as transições. Quando tentamos descrever o código em uma máquina de estado simples, conforme descrito aqui, pode ficar claro que nenhum diagrama captura todas as interações e o código pode seguir caminhos diferentes em situações diferentes. Isso pode acontecer se os desenvolvedores não pensaram nesse código como uma máquina de estado e não o implementaram de forma estruturada. Se for esse o caso, considere atrasar o recurso para que ele possa

ser feito corretamente; caso contrário, a complexidade de uma máquina de estado mal implementada pode introduzir muitos *bugs*.

Para as diferentes máquinas de estado, considere estas questões:

1. Qual é o estado inicial?
2. Existem diferentes configurações possíveis mesmo desde o início da máquina de estado?
3. Quais são todos os pontos de entrada do estado?
4. Por exemplo, se podemos criar um usuário a partir de uma página da *web*, uma chamada de API ou um aplicativo, adicione um teste para cada.
5. Existem estados que retornam a estados anteriores?
6. Existem modos de falha que o movem entre os estados?
7. Quais são todos os pontos de saída de estado, incluindo modos de falha?
8. Algum estado está realmente separado e se comporta de maneira diferente com base em uma variável ou ambiente?
9. Podemos precisar dividir um estado se ele tiver modos muito diferentes de trabalhar em situações diferentes, especialmente se suas entradas e saídas forem diferentes.
10. Identificamos todas as transições e adicionou um teste para cada uma delas?
11. Exceto nos casos mais simples, a verificação de cada transição requer pelo menos tantos testes quanto a verificação de todos os estados.
12. Que sequências de transições podem interagir umas com as outras?
13. Quais são os estados finais?

Podemos gerar muitos testes aplicando essas perguntas à máquina de estado de seu aplicativo, conforme demonstrado no exemplo da Figura 2.

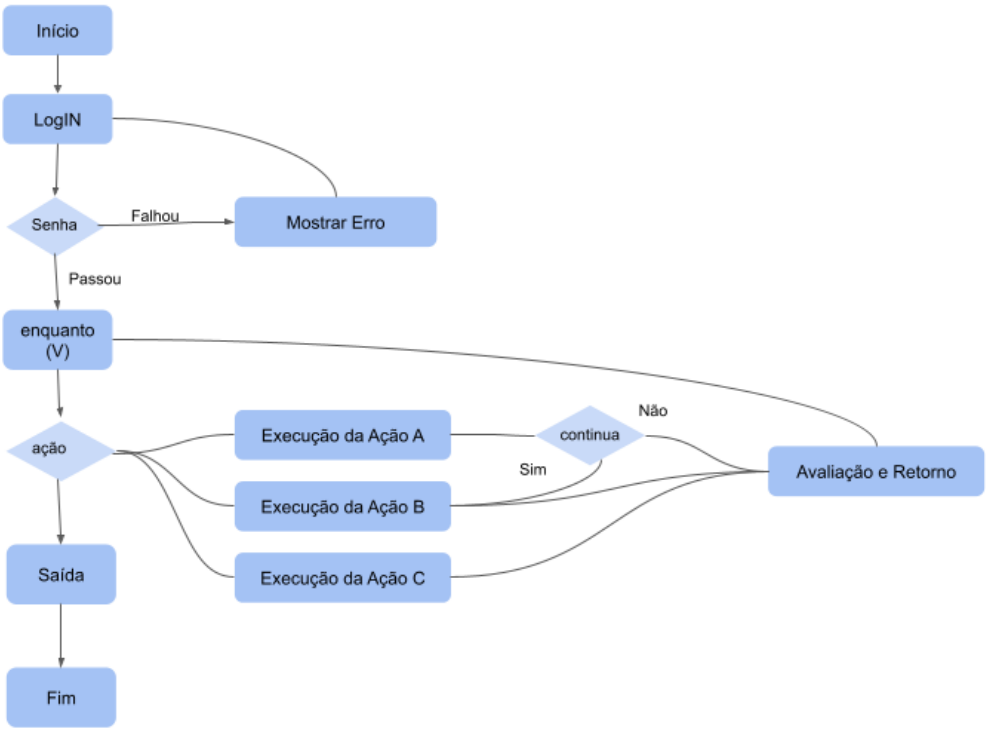
Nesse aplicativo simples, os usuários fazem *login* e selecionam entre quatro opções: ação A, ação B, ação C ou sair. Até que eles saiam, o programa faz um *loop* para que eles possam escolher outra ação. As ações A e B estão vinculadas para que, após executar a ação A, possamos continuar com a ação B.

Com esses quatro testes, podemos verificar todos os estados e transições. Procure instâncias em seu aplicativo quando a cobertura do estado não for suficiente para fornecer cobertura de transição. Priorize seus testes para

que as transições importantes tenham testes e, se optarmos por não testar todas, certifique-se de pelo menos identificá-las para poder tomar uma decisão informada.

Mesmo com todas as transições testadas, ainda pode haver problemas dependendo da combinação de estados pelos quais o aplicativo passa.

Figura 2 – Exemplo de estado de máquina



Cobertura de N-switch: agora, testamos todos os estados e transições. No entanto, este programa inclui dois *loops*. O resultado de um *loop* pode afetar execuções subsequentes. Se uma falha de *login* simplesmente adiciona uma mensagem de erro à página, por exemplo, duas falhas de *login* seguidas podem mostrar a mensagem de erro duas vezes. Ou se a ação A for inválida após a execução da ação B, esses testes não encontrarão esse erro.

As sequências de estados que visitamos. Se essas sequências tiverem um estado de comprimento, teremos cobertura de 0 switch. Considerando uma série de *n* estados juntos, temos cobertura de switch n-1. Para obter cobertura de 1 chave e encontrar erros em sequências de dois estados em nosso programa de exemplo, precisamos de casos de teste adicionais.

O teste 1 abrange o caso de várias falhas de *login*. Os testes 1 a 3 abrangem os casos de falha de *login* seguidos por cada uma das ações executadas uma vez. Os testes 4 a 12 tentam o caso de *login* bem-sucedido

seguido por todas as permutações de pares de ações A, B e C em sequência. Se houver um *bug* ao executar a ação B, então a ação A, este nível de teste, irá encontrá-lo.

Obviamente, com um *loop* infinito, podemos estender isso indefinidamente por meio de cobertura de 2 comutadores, 3 comutadores e assim por diante. Eles descobrem *bugs* que são mais obscuros e menos prováveis de serem atingidos, para que possamos escolher o nível que deseja alcançar. Um princípio de teste vital é sempre estar ciente do que não estamos cobrindo, mesmo que optemos por não gastar tempo testando.

Algumas combinações de transições serão relacionadas e outras serão independentes. Consideramos várias falhas de *login* e diferentes combinações de ações, e ambas podem ocultar *bugs*. Mas e se tivermos uma falha de *login* antes de executar as ações A, B e C? Para cobertura de 1 *switch*, devemos incluí-los também. No entanto, essas duas partes do sistema não estão relacionadas: depois de fazer *login*, não importa quantas falhas de *login* tivemos anteriormente, podemos podar sua lista de possíveis casos de teste identificando áreas não relacionadas que não estão vinculadas na prática.

Nesse caso, há tantas combinações de ações, A, B e C, que precisaremos fazer *login* várias vezes. Nesse caso, podemos adicionar variedade ao *login*, pois executará vários outros testes. Procure outras oportunidades para adicionar variação. Se uma área do código for particularmente complexa e exigir muitos testes para tentar todos os seus caminhos, não execute todos esses caminhos com o mesmo nome de usuário, por exemplo. Como estamos executando esses testes de qualquer maneira, tente-os com muitos nomes de usuário diferentes, caso haja uma interação.

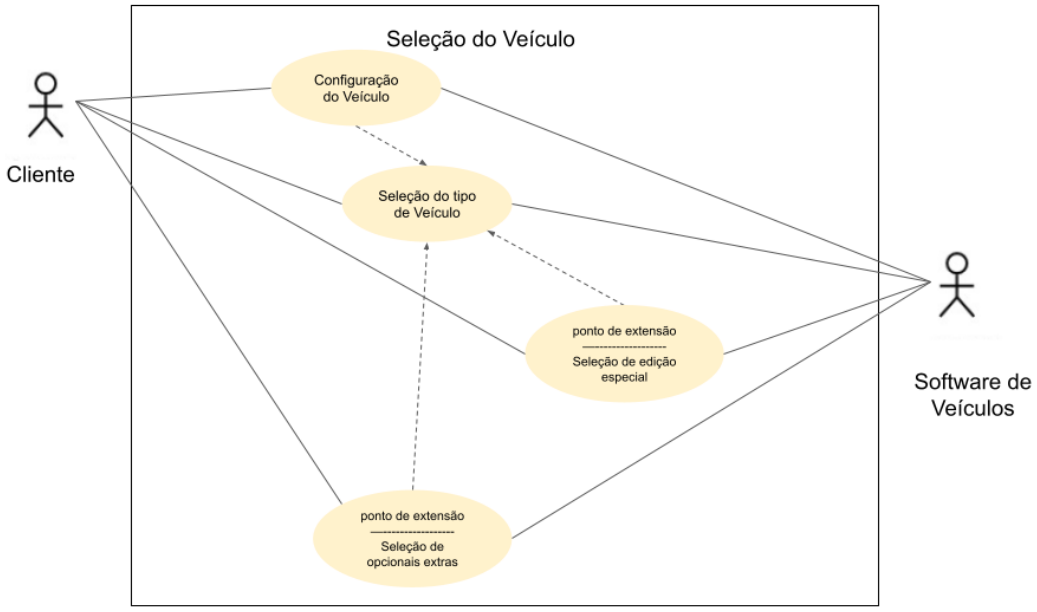
Após cada transição, precisamos verificar a condição do sistema. Para máquinas de estado com uma variável explícita rastreando seu estado, devemos começar verificando essa variável. Em seguida, considere as possibilidades de entrada e saída. Isso pode significar que determinados botões ou elementos da interface do usuário estão visíveis para permitir a transição para a próxima etapa ou que o aplicativo responde a mensagens ou entradas específicas da API. Calcule tudo o que distingue cada estado do próximo para que possamos verificar todos eles.

Alguns programas são complexos demais para serem descritos com uma máquina de estado único e requerem uma abordagem diferente para o teste.

3.3. Teste de caso de uso

Casos de uso e casos de negócios são frequentemente usados para determinar e documentar os requisitos do sistema. Estes são geralmente ilustrados usando diagramas de caso de uso. Esses diagramas descrevem interações típicas de uso/sistema e são usados para especificar requisitos em um nível bastante abstrato. A Figura 3 mostra um diagrama de caso de uso para parte do processo de seleção de um veículo.

Figura 3 – Diagrama de caso de uso para seleção de veículo



Fonte: elaborado com base em Spillner, 2007.

Os casos de uso individuais neste exemplo são “configurar veículo”, “selecionar tipo de veículo”, “selecionar edição especial” e “selecionar extras opcionais”. As relações entre eles são classificadas usando as *tags* “extend” e “include”. Os relacionamentos “incluir” sempre ocorrem, enquanto os relacionamentos “estender” ocorrem apenas em circunstâncias específicas em “pontos de extensão”. Em outras palavras, existem alternativas para um relacionamento de “estender” ou o relacionamento simplesmente não entra em vigor.

O diagrama representa a seguinte situação: para configurar um veículo, o cliente deve selecionar um tipo de veículo. Uma vez que isso aconteceu, existem três maneiras alternativas de proceder. O cliente pode selecionar uma edição

especial, extras opcionais ou nenhum deles. O sistema de veículos está envolvido em todas as três ações, que requerem seus próprios diagramas de casos de uso detalhados.

Os diagramas de caso de uso geralmente descrevem a visão externa de um sistema e servem para esclarecer a visão do usuário sobre o sistema e suas relações com outros sistemas conectados. Linhas e desenhos simples, como os bonecos no diagrama acima, indicam relacionamentos com entidades externas, como pessoas ou outros sistemas.

Cada caso de uso define um comportamento específico que um objeto pode executar em colaboração com uma ou mais outras entidades. Os casos de uso são descritos usando interações e atividades que podem ser aumentadas com pré e pós-condições. A linguagem natural também pode ser usada para estender essas descrições na forma de comentários ou acréscimos a cenários individuais e suas alternativas. As interações entre entidades externas e o objeto podem levar a uma mudança no estado do objeto, e as próprias interações podem ser ilustradas em detalhes usando fluxos de trabalho, diagramas de atividades ou modelos de processos de negócios.

- Pré e pós-condições: cada caso de uso está sujeito a pré e pós-condições específicas que precisam ser atendidas para que o caso de uso seja executado com sucesso. Por exemplo, uma das pré-condições para a configuração de um veículo é que o cliente esteja conectado ao sistema. As pós-condições entram em jogo, uma vez que o caso de uso foi executado. Por exemplo, uma vez que um veículo foi selecionado com sucesso, a configuração pode ser solicitada *on-line*. A sequência de casos de uso dentro de um diagrama (ou seja, o “caminho” percorrido) também depende de pré e pós-condições.
- Adequado para testes de sistema e aceitação: os casos de uso e os diagramas de casos de uso servem como base de teste ao projetar testes baseados em casos de uso. Como esse tipo de teste modela uma visão externa do sistema, ele é altamente adequado para testes de sistema e de aceitação. Se um diagrama modela a interação e as dependências entre os componentes individuais do sistema, ele também pode ser usado para derivar casos de teste de integração.
- Testando o uso “normal” do sistema: os diagramas de casos de uso ilustram a sequência “usual” ou mais provável de eventos e suas

alternativas, de modo que os testes derivados deles são usados para verificar cenários típicos de uso do sistema. Para que um sistema seja aceito, é importante que funcione sem erros em condições “normais”. Isso torna os testes baseados em casos de uso extremamente importantes para o cliente e, portanto, também para desenvolvedores e testadores.

Um caso de uso geralmente compreende várias variantes de seu comportamento básico. No nosso exemplo, uma variante é a seleção de uma edição especial, que por sua vez impossibilita o cliente de selecionar quaisquer outros extras opcionais. Casos de teste mais detalhados também podem ser usados para modelar casos especiais e tratamento de erros.

- Casos de teste: cada caso de uso está relacionado a uma tarefa específica e a um resultado específico esperado. Podem ocorrer eventos que levam a outras atividades ou ações alternativas e as pós-condições estão presentes após a execução. Para projetar um caso de teste, precisamos saber:
  - a situação inicial e as pré-condições exigidas;
  - quaisquer restrições relevantes;
  - os resultados esperados;
  - as pós-condições necessárias.

No entanto, valores de entrada específicos e resultados para casos de teste individuais não podem ser derivados diretamente de casos de uso. Cada caso de teste deve ser desenvolvido com dados apropriados. Todos os cenários alternativos mostrados no diagrama de caso de uso (ou seja, os relacionamentos de “extensão”) também devem ser cobertos por casos de teste individuais. Os casos de teste projetados com base em cenários de caso de uso podem ser combinados com outras técnicas de teste baseadas em especificação.

- Definindo Critérios de Saída: um possível critério de saída é que cada caso de uso (ou sequência de casos de uso) no diagrama seja coberto por pelo menos um caso de teste. Como os caminhos alternativos e/ou extensões também são casos de uso, esse critério exige que cada alternativa/extensão seja executada.

O grau de cobertura pode ser medido dividindo o número de variantes de caso de uso que você realmente testa pelo número total de casos de uso

disponíveis. Esse grau de cobertura geralmente é expresso como uma porcentagem.

- Benefícios e Limitações: são ideais para testar interações típicas de usuário/sistema, tornando-os ideais para aceitação e teste de sistema. Exceções “previsíveis” e casos especiais podem ser ilustrados no diagrama de casos de uso e podem ser cobertos por casos de teste adicionais. No entanto, não há uma maneira simples e metódica de derivar outros casos de teste que abranjam situações que estão além do escopo do diagrama. Para situações como essa, precisamos usar outras técnicas, como análise de valor de contorno.

Com isso, encerramos o conteúdo de testes de casos de uso, sabendo que há um caminho longo para experimentarmos e aperfeiçoarmos esse tipo de teste.

TEMA 4 – TESTE DE CAIXA BRANCA: TESTES DE INSTRUÇÕES E COBERTURA, TESTES DE DECISÃO E COBERTURA

As técnicas de teste de caixa branca são baseadas na estrutura interna do objeto de teste e geralmente são chamadas de *testes estruturais* ou *baseados em estrutura*. Outro termo para essa técnica é *teste relacionado a código*. Esse tipo de teste depende da disponibilidade do código-fonte e da capacidade de manipulá-lo e adaptá-lo, se necessário. As técnicas de teste de caixa branca podem ser usadas em todos os níveis de teste, mas as técnicas descritas abaixo são voltadas diretamente para o nível de teste de componentes (Spillner, 2007).

Juntamente a essas técnicas existem muitas outras que visam alcançar um maior grau de cobertura de código e, portanto, aumentar a eficácia do teste. Essas técnicas geralmente são usadas para testar sistemas críticos de segurança ou de negócios.

O conceito principal de um teste do tipo caixa branca é garantir que todas as partes do código do objeto de teste sejam executadas pelo menos uma vez. Os casos de teste baseados em processo são projetados e executados com base na lógica do programa. É claro que as especificações do objeto também desempenham um papel no projeto de casos de teste e na decisão se um comportamento defeituoso foi detectado quando um teste é executado.

O objetivo é atingir um grau predefinido de cobertura das declarações durante o teste, ou seja, obter o maior número possível de declarações no programa para executar o que for possível. Podemos diferenciar entre os seguintes tipos de técnicas de teste de caixa branca:

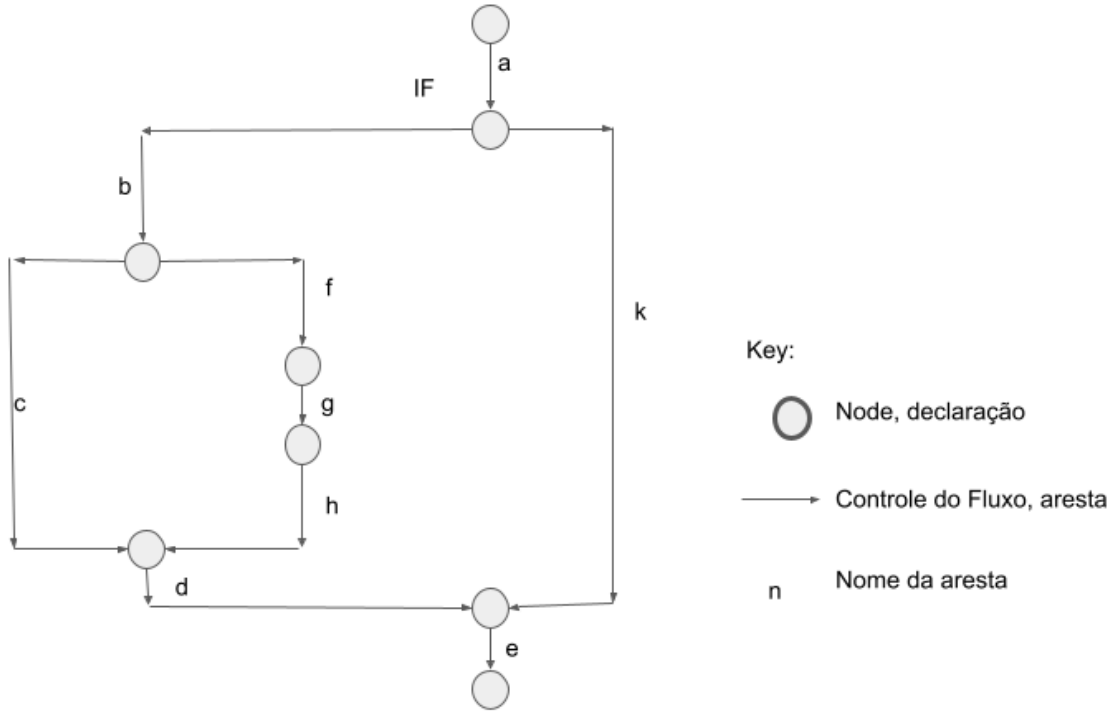
- Teste de Declaração;
- Teste de Decisão;
- Teste de Condição;
- Teste de Condição de Ramificação;
- Teste de Combinação de Condição de Ramificação;
- Teste de Cobertura de Decisão de Condição Modificada;
- Teste de Caminho.

4.1 Teste e cobertura da declaração

As declarações individuais do objeto de teste são o foco desse tipo de teste. Você precisa identificar os casos de teste que executam uma porcentagem predefinida (ou todas) das instruções executáveis do programa. Essa técnica (e o teste de decisão) pode ser ilustrada e explicada usando um gráfico de fluxo de controle, que pode ser criado com base no código-fonte.

A Figura 4 ilustra os fluxos possíveis dentro do objeto de teste e permite definir com precisão a cobertura necessária. As instruções são representadas por nós e o fluxo de controle por arestas direcionadas que conectam os nós. Se o programa contém uma sequência de instruções incondicionais, isso é representado por apenas um nó, pois a execução da instrução inicial leva à execução de todas as instruções subsequentes também. As instruções de seleção (IF, CASE) e loops (WHILE, FOR) têm várias arestas de saída.

Figura 4 – Gráfico de fluxo de controle para um componente do sistema



Fonte: elaborado com base em Spillner, 2007.

Após a execução dos casos de teste, precisamos verificar quais instruções individuais foram executadas. Se o grau de cobertura exigido for alcançado, o teste é considerado completo. Geralmente, executaremos todas (100%) as instruções, pois é impossível julgar se uma instrução que não foi executada funciona conforme o esperado.

4.2 Casos de teste (cobertura de nós no diagrama de fluxo de controle)

Em nosso exemplo, todas as instruções (ou seja, todos os nós) podem ser alcançadas usando um único caso de teste. Este caso de teste deve percorrer todas as arestas na seguinte ordem:

a, b, f, g, h, d, e

Depois que todas as arestas forem percorridas, todas as instruções (nós) serão executadas uma vez. Certamente, haverá outras sequências que também fornecem 100% de cobertura. Lembre-se, porém, que minimizar o esforço geral é um dos princípios orientadores do teste, ou seja, devemos cumprir o grau predefinido de cobertura com o menor número possível de casos de teste.

O comportamento esperado do programa e os resultados esperados devem ser definidos com base nas especificações e, uma vez executado o teste, é necessário comparar os resultados esperados com os resultados reais para identificar eventuais divergências ou falhas.

4.3 Definindo critérios de saída

Os critérios de saída para este tipo de teste podem ser claramente definidos usando a seguinte fórmula:

Cobertura da declaração = (número de instruções executadas / número total de instruções)×100%

4.4 Cobertura C0

Essa medida para calcular a cobertura da instrução também é conhecida como *cobertura C0* (C-zero) e fornece apenas um critério de saída fraco. Atingir 100% de cobertura de instrução pode ser difícil, por exemplo, se o programa incluir exceções que são difíceis de produzir durante o teste sem gastar muito esforço extra.

4.5 Benefícios e limitações (Identificando código morto)

Se 100% de cobertura for necessária, mas algumas instruções não puderem ser executadas por nenhum caso de teste, isso pode ser um sinal de código inacessível ou “morto”.

4.6 Arestas ELSE vazias são ignoradas

Se uma condição (instrução IF) só executa instruções depois de cumprida (a parte THEN) e não há parte ELSE, o gráfico de fluxo de controle tem uma borda THEN com pelo menos um nó começando na condição, mais um segundo ELSE de saída *edge* sem nenhum nó intermediário (sem instruções na parte ELSE). O fluxo de controle dessas duas arestas é reunido no nó terminal (ENDIF) da instrução IF. Uma borda ELSE vazia (entre IF e ENDIF) é irrelevante ao calcular a cobertura da instrução e quaisquer instruções ausentes nesta parte do programa não serão detectadas usando o teste de instrução.

4.7 Teste de decisão e cobertura (mais casos de teste necessários)

Um critério mais avançado para o teste de caixa branca é o teste de decisão e cobertura, em que as decisões dentro do código-fonte são o foco do processo de teste. Nesse caso, o efeito de uma decisão é avaliado e, como resultado, é tomada uma decisão sobre qual(is) instrução(ões) executar(em) a seguir. Esta situação deve ser considerada durante o teste. As decisões no código são acionadas por instruções IF e CASE, *loops* e outras instruções semelhantes. A consulta implícita em uma instrução IF é analisada e um resultado booleano “verdadeiro” ou “falso” é retornado (ou seja, um IF é uma decisão com dois resultados possíveis). Uma instrução CASE (ou SWITCH) tem um resultado para cada opção disponível, incluindo a opção padrão.

Os testes de decisão também podem ser ilustrados usando gráficos de fluxo de controle, mas são referidos como teste de ramificação com cobertura de ramificação correspondente. As arestas individuais no grafo (ou seja, as “ramificações” entre os nós) são o foco do procedimento de teste.

4.8 Casos de teste adicionais necessários

Vamos começar com um exemplo para um teste de ramificação. Para nosso exemplo (veja a Figura 4), precisamos de casos de teste adicionais se quisermos cobrir todas as ramificações no gráfico de fluxo de controle. A sequência de arestas a seguir nos deu 100% de cobertura de instrução:

a, b, f, g, h, d, e

No entanto, as arestas c, i e k não são executadas por este caso de teste. As arestas *c* e *k* são resultados “falsos” de uma instrução IF, e a aresta *i* é o salto de retorno ao início do *loop*. Isso significa que precisamos de um total de três casos de teste para gerar 100% de cobertura de ramificação:

a, b, c, d, e

a, b, f, g, eu, g, h, d, e

a, k, e

4.9 Cobertura do resultado da decisão no código

Expresso na forma de decisões (somente o único caso de teste com 100% de cobertura de instrução): para a primeira instrução IF, falta um resultado da decisão (a aresta *k* não é executada durante o teste). O mesmo se aplica à segunda instrução IF (a aresta *c* também não é testada). Além disso, uma repetição do *loop* WHILE (a borda *i*) ainda não foi testada.

4.10 Cobertura de borda no gráfico de fluxo de controle

Juntos, os três casos de teste fornecem cobertura completa de todas as arestas em nosso gráfico de fluxo de controle (ou seja, todas as ramificações possíveis do fluxo de controle no código são cobertas por pelo menos um caso de teste). Da mesma forma, todas as decisões — ou, mais precisamente, todos os resultados das decisões — são cobertos por esses três casos de teste.

Como em nosso exemplo, muitas vezes é impossível evitar a execução de algumas arestas várias vezes. Aqui, como não há caminhos alternativos, as arestas *a* e *e* são executadas por todos os três casos de teste.

Aqui também, as pré e pós-condições, os resultados esperados e o comportamento esperado do objeto de teste devem ser definidos antecipadamente e comparados com os resultados do teste. Para ajudar a identificar discrepâncias de tempo de execução, também é útil registrar quais resultados de decisão são entregues e/ou quais ramificações são seguidas durante o teste. Isso é especialmente importante quando se trata de identificar instruções ausentes em ramificações vazias.

4.11 Definindo critérios de saída

Analogamente à cobertura de declaração, a cobertura de decisão é calculada da forma abaixo, considerando que:

- NRT = número de resultados de teste durante o teste;
- TDP = número total de decisões possíveis resultados no objeto de teste;
- Cobertura de decisão = (NRT / TDP) x 100%.

4.12 Cobertura C1

A cobertura de decisão é chamada de *cobertura C1*. O cálculo apenas analisa se um resultado/ramificação de decisão é percorrido, não com que frequência. Em nosso exemplo, os ramos *a* e *e* (e os resultados de decisão correspondentes) são percorridos três vezes, uma vez para cada caso de teste.

Ao medir a cobertura, lembre-se de que os *loops* são executados várias vezes. Para componentes críticos, você precisa usar técnicas especializadas, como teste de limites internos.

Geralmente, não é possível atingir 100% de cobertura de caminho em um objeto de teste. A cobertura do caminho é considerada uma métrica teórica e, devido ao esforço excessivamente alto envolvido, tem pouco ou nenhum significado na prática.

As técnicas de caixa branca são mais bem aplicadas a testes de baixo nível, enquanto as técnicas de caixa preta podem ser aplicadas em todos os níveis de teste.

4.13 Utilize sempre a experiência

É sempre benéfico complementar as técnicas de teste de caixa branca usando técnicas de teste intuitivas, baseadas em lista de verificação e/ou exploratórias que aproveitam a experiência de seus testadores. Esses tipos de testes geralmente revelam outros defeitos que tais testes ignoram.

TEMA 5 – TESTES BASEADOS NA EXPERIÊNCIA E INTUIÇÃO

5.1 Cobertura apenas parcialmente ou nada mensurável

Juntamente às técnicas metódicas de derivação de casos de teste, os testadores também podem usar sua experiência e intuição para derivar casos de teste adicionais. O teste baseado em experiência pode ser usado para descobrir defeitos que muitas vezes são ignorados pelo teste sistemático. Geralmente, é prudente aumentar os testes sistemáticos com testes baseados na experiência. Esses tipos de técnicas podem variar em sua eficácia na descoberta de defeitos, dependendo das áreas de especialização do testador. O teste baseado na experiência torna difícil (ou simplesmente impossível) definir um grau de cobertura que possa ser usado como critério de saída (Spillner, 2007).

Além da experiência de teste, os testadores podem usar sua experiência no desenvolvimento de aplicativos semelhantes ou conhecimento das tecnologias que estão sendo usadas para derivar casos de teste apropriados. Por exemplo, um testador com experiência em uma determinada linguagem de programação pode usar defeitos cujas causas estão no uso da linguagem como base para derivar casos de teste para o projeto atual.

5.2 Derivação intuitiva de casos de teste (erro ao adivinhar)

O *design* de caso de teste intuitivo é baseado na capacidade intuitiva de usar erros, falhas e falhas esperados para selecionar casos de teste adequados. Nesse caso, não há técnica metódica, e o testador deve confiar na capacidade de reconhecer situações em que ocorreram defeitos no passado (e em outras aplicações), ou no conhecimento de como funcionavam as versões anteriores do objeto de teste, ou na consciência de quais tipos de erros os desenvolvedores provavelmente cometerão. Essa técnica de derivação de caso de teste comumente usada é chamada *de suposição de erro*.

5.3 Registre suas experiências em listas de verificação

Uma maneira metódica de apoiar esta técnica é manter listas de erros, falhas e falhas que você descobriu no decorrer de projetos anteriores, ou que você compilou a partir de seu próprio conhecimento das razões pelas quais o *software* falha, e usá-los para ajudar a derivar novos casos de teste.

Tente sempre manter essas listas atualizadas e atualize-as com novas experiências assim que elas ocorrerem. Você pode determinar um tipo de figura de cobertura calculando a porcentagem de entradas de lista que você usa para escrever casos de teste.

5.4 Teste baseado em lista de verificação

Como o nome sugere, essa técnica usa listas de verificação como base para derivar casos de teste. Uma lista de verificação contém aspectos de um programa (ou seja, condições de teste) que precisam ser testados — isso inclui coisas como tratamento de exceções e confiabilidade. As entradas em uma lista de verificação devem lembrar o testador de incluir esses aspectos nos casos de teste resultantes. Outros aspectos incluem regulamentos, casos especiais ou

condições de dados que requerem verificação. O nível de detalhamento das listas de verificação varia enormemente de projeto para projeto.

5.5 Mantenha suas listas de verificação atualizadas

As listas de verificação são amplamente baseadas na experiência de um testador individual. Eles geralmente são compilados por longos períodos de tempo e são atualizados regularmente. No entanto, isso não significa que você não deva usar uma lista de verificação padronizada e não editada.

Podemos usar diferentes listas de verificação para diferentes tipos de teste — por exemplo, uma lista pode conter condições para testes funcionais, enquanto outra contém as condições para testes não funcionais.

O uso de listas de verificação pode ajudá-lo a manter um grau de consistência durante o teste. No entanto, como as condições que eles contêm são redigidas de maneira geral, os casos de teste derivados de seu uso variam, dependendo de qual testador acaba testando quais condições. Essa variação aumenta a cobertura dos aspectos individuais na lista, mas reduz o nível geral de repetibilidade do teste.

5.6 Cobertura

Podemos medir a cobertura calculando a porcentagem de entradas de lista que são cobertas por casos de teste. Se todos os aspectos da lista forem testados, alcançamos 100% de cobertura e pode interromper o teste.

FINALIZANDO

Apresentamos particularmente características e formas de uso dos testes tipo caixa preta e caixa branca.

Embora os testes de caixa branca se fixem na estrutura do código, eles não são mais importantes que o de caixa preta que observa apenas o funcionamento sem a importância de conhecimento do código.

O que nos importa é sabermos que ambos são complementares, e que o de caixa branca geralmente vem mais próximo ao momento do desenvolvimento, enquanto o de caixa preta segue o caminho das validações conforme as entregas vão sendo feitas.

Há um número grande de subitens em relação às duas técnicas, mas nem sempre conseguimos utilizar o conjunto todo de técnicas. É importante que consigamos adaptar as técnicas essenciais e necessárias a cada tipo de projeto ou empresa. Testes de *software* são de fato caros para as empresas quando pensamos na estrutura de pessoas, tempo utilizado, entre outros custos, porém, o seu retorno em qualidade e confiabilidade pelos clientes pode custar muito mais caro se negligenciarmos estes processos.

REFERÊNCIAS

AMEY, S. ***Software*** **Test** ***Design***. Packt Publishing, 2022.

ANICHE, M. **Effective** ***Software*** **Testing**. Manning Publications, 2022.

ASSOCIATION FOR COMPUTING MACHINERY. Disponível em: <https://www.acm.org/>. Acesso em: 12 mar. 2023.

BECK, K. **Test-driven development**. Addison-Wesley, 2003.

BROWN, E. **Web development with Node and Express**. Sebastopol: O'Reilly Media, Inc., 2014.

DESIKAN, S. ***Software*** **Testing**: Principles and Practices. O'Reilly Media, Inc., 2007.

GARCIA, B. **Mastering** ***Software*** **Testing with JUnit 5**. Packt, 2017.

HAMBLING, B. et al. ***Software*** **testing**: an ISTQB-BCS certified tester foundation guide. 3. ed. Swindon: BCS Learning & Development Ltd., 2015.

IBM100. **The Origins of Computer Science**. Disponível em: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/compsci/>. Acesso em: 12 mar. 2023.

IEEE – Institute of Electrical and Electronic Engineers. **The world's largest technical professional organization dedicated to advancing technology for the benefit of humanity**. Disponível em: <https://www.ieee.org/>. Acesso em: 12 mar. 2023.

ISO – International Organization for Standardization. **ISO 8402**:1994. Disponível em: <https://www.iso.org/standard/20115.html>. Acesso em: 12 mar. 2023.

_____. **ISO 9001 and related standards — Quality management**. Disponível em: <https://www.iso.org/iso-9001-quality-management.html>. Acesso em: 12 mar. 2023.

_____. **ISO/IEC 25010**:2011. Disponível em: <https://www.iso.org/standard/35733.html>. Acesso em: 12 mar. 2023.

_____. **ISO/IEC/IEEE 24765**:2017. Disponível em: <https://www.iso.org/standard/71952.html>. Acesso em: 12 mar. 2023.

ISTQB – International *Software* Testing Qualifications Board. Certified Tester **Foundation Level (CTFL)**. Disponível em: <https://www.istqb.org/certifications/certified-tester-foundation-level>. Acesso em: 12 mar. 2023.

JUNIT. **Teste de Unidade**. Disponível em: <http://junit.wikidot.com/>. Acesso em: 12 mar. 2023.

LAPORTE, C.; APRIL, A. ***Software* quality assurance**. Wiley: IEEE Press, 2018.

LATINO, R. **Root cause analysis**: improving performance for bottom-line results. CRC Press, 2011.

LEWIS, W. E. ***Software* testing and continuous quality improvement**. 3. ed. Boca Raton: Taylor & Francis Group, LLC., 2009.

MOHAN, G. **Full Stack Testing**. O'Reilly Media, Inc., 2022.

PRESSMAN, R. S. **Engenharia de *software***: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de *software***. São Paulo: Pearson Education do Brasil, 2018.